

# Research - Kasteran\* — Automatic Differentiation

Alpasan, Lois-Kleinner

2026

## Abstract

Automatic differentiation (AD) is a set of techniques for computing derivatives of functions expressed as computer programs. Unlike numerical differentiation (finite differences) or symbolic differentiation, AD produces exact derivatives to machine precision by systematically applying the chain rule of calculus. This document surveys forward-mode and reverse-mode AD, the Wengert tape (trace) representation, dual numbers, and the implementation of AD in machine learning frameworks (TensorFlow, PyTorch, DiffSharp). We discuss Kasteran\*'s built-in AD system, which leverages the linear type system for efficient gradient computation.

## Kasteran\* — Automatic Differentiation

Academic Research Document © Lois-Kleinner & 0-1.gg 2026

### Abstract

Automatic differentiation (AD) is a set of techniques for computing derivatives of functions expressed as computer programs. Unlike numerical differentiation (finite differences) or symbolic differentiation, AD produces exact derivatives to machine precision by systematically applying the chain rule of calculus. This document surveys forward-mode and reverse-mode AD, the Wengert tape (trace) representation, dual numbers, and the implementation of AD in machine learning frameworks (TensorFlow, PyTorch, DiffSharp). We discuss Kasteran\*'s built-in AD system, which leverages the linear type system for efficient gradient computation.

## 1. Introduction

Differentiation is a fundamental operation in scientific computing, optimization, and machine learning. Gradient-based optimization—using derivatives to find minima or maxima of functions—underpins training of neural networks, parameter estimation in physical models, and optimal control. Automatic differentiation provides a principled, efficient way to compute derivatives of arbitrary computations, including those with control flow, iteration, and recursion. Kasteran\* includes a first-class AD system that automatically differentiates any function annotated with the `[grad]` attribute, generating both the primal computation and its derivative.

## 2. Historical Background

The origins of automatic differentiation trace back to the 1950s and 1960s, when researchers in numerical analysis recognized that the chain rule could be applied systematically to computer programs (Bücker et al. 1). The Wengert tape, introduced by Wengert in 1964, provided the first

explicit formulation of reverse-mode AD: record the sequence of elementary operations during a forward pass, then propagate derivatives backward through the recorded trace (Wengert 1). This “backpropagation” became the foundation of neural network training.

The “fast automatic differentiation” work of Baur and Strassen established theoretical bounds on the complexity of derivative computation (Baur and Strassen 1). They proved that the gradient of a function with  $n$  inputs and  $m$  operations can be computed in  $O(m)$  time—the same asymptotic complexity as the original function—using reverse-mode AD. This result, known as the “cheap gradient principle,” explains why reverse-mode AD is the method of choice for optimization problems with many inputs and few outputs.

The development of dual numbers provided an elegant mathematical framework for forward-mode AD. A dual number is an element of the ring  $R[\epsilon]/(\epsilon^2)$ : a number of the form  $a + b\epsilon$ , where  $\epsilon^2 = 0$ . Computing a function  $f(a + b\epsilon)$  yields  $f(a) + f'(a)b\epsilon$  through the formal algebra of dual numbers, without any tracing or tape recording (Clifford 1). Forward-mode AD uses dual numbers to compute directional derivatives in a single forward pass.

The modern era of AD began with the integration of reverse-mode AD into machine learning frameworks. TensorFlow introduced “define-and-run” computation graph construction with symbolic differentiation (Abadi et al. 1). PyTorch’s “define-by-run” approach builds the computation graph dynamically during execution, enabling imperative-style programming with automatic gradient computation (Paszke et al. 1). DiffSharp, a functional AD library for F#, demonstrated that AD can be integrated into a statically-typed, functional language with algebraic effects (Baydin et al. 1).

### 3. Technical Analysis

Kasteran\*’s AD system supports both forward and reverse mode, selected through a pragma annotation. The system is implemented as a compiler pass operating on the HIR, before lowering to LLVM IR.

Forward-mode AD replaces each scalar value with a dual number (value, derivative). The compiler transforms operations by applying the chain rule:

```
Original:    y = f(x)           where f: R → R
Transformed: y = f(x)           (value)
              dy = f'(x) * dx    (derivative)
```

For vector-valued functions, forward-mode computes a Jacobian-vector product (JVP):

```
y = f(x)
dy = J_f(x) · v    // JVP, where v is the seed vector
```

The transformation is straightforward for elementary operations:

```
z = x + y      →    dz = dx + dy
z = x * y      →    dz = x * dy + y * dx
z = sin(x)     →    dz = cos(x) * dx
z = if c then a else b
              →    dz = if c then da else db
```

Reverse-mode AD records a trace of operations during the forward pass, then differentiates backward. The compiler generates two functions: a forward function that computes the result and

records the trace, and a backward (adjoint) function that propagates gradients:

```
// Forward pass: compute y = f(x)
// Let v_i be intermediate values, recorded in the tape T

fn f_forward(x) {
  let mut tape = Tape::new();
  let v0 = x;
  let v1 = v0 * v0;  tape.record(Mul(v0, v1));
  let v2 = sin(v1);  tape.record(Sin(v1, v2));
  let y = exp(v2);   tape.record(Exp(v2, y));
  (y, tape)
}

// Reverse pass: propagate gradients through the tape

fn f_backward(tape, gy) {
  let mut partials = HashMap::new();
  partials[y] = gy;
  for (op, inputs, output) in tape.reverse() {
    match op {
      Mul(x, z) => { partials[x] += z * partials[z]; partials[z] += x * partials[z]; }
      Sin(x)    => { partials[x] += cos(x) * partials[output]; }
      Exp(x)    => { partials[x] += exp(x) * partials[output]; }
    }
  }
  partials[x]
}
```

The Wengert tape is a linear data structure that captures the data dependencies between operations. Each entry records the operation type, the input values, and the output value. The tape can be optimized through “checkpointing” (discarding intermediate values that can be recomputed) and “taping to file” for out-of-core computation on large graphs.

Kasteran\*’s AD system leverages the linear type system for efficient gradient computation. Because linear values cannot be aliased, the compiler can safely in-place update gradient accumulators without atomic operations. The capability system tracks which gradient buffers are writable, enabling automatic parallelization of gradient computation across multiple threads.

#### 4. Current State of the Art

The AD landscape is dominated by machine learning frameworks. TensorFlow’s gradient tape records operations on eager tensors and uses the recorded trace for automatic differentiation. JAX, a functional AD framework for Python, provides both JVP and VJP (vector-Jacobian product, the reverse-mode primitive) with composition through function transformations (Bradbury et al. 1). JAX uses XLA for just-in-time compilation of gradient computations to GPU and TPU.

PyTorch’s autograd engine builds a dynamic computation graph during forward execution, recording each operation in a directed acyclic graph. The gradient computation traverses the graph in topological order, propagating gradients from outputs to inputs. PyTorch supports custom au-

tograd functions, gradient checkpointing, and distributed gradient computation across multiple GPUs.

DiffSharp, an AD library for F#, demonstrates AD in a purely functional setting using algebraic effects and differentiable programming. It supports both forward and reverse modes, higher-order derivatives, and nested differentiation. The design shows that AD can be integrated naturally into a language with statically checked effects and algebraic type systems.

## 5. Relevance to Kasteran\*

Kasteran\*'s AD system is integrated into the compiler, not implemented as a library. This enables optimizations that are impossible in library-based AD systems: (1) the compiler can inline operations and eliminate tape entries through constant folding and dead code elimination, (2) the linear type system ensures that gradient accumulators are not aliased, enabling safe in-place updates, and (3) the capability system tracks gradient dependencies, enabling automatic parallelization of backpropagation.

The AD system supports nested differentiation (Hessian computation) through repeated application of the forward/reverse transformations. Higher-order derivatives are useful in optimization algorithms (Newton's method), physics simulation (variational integrators), and sensitivity analysis.

Kasteran\* programs can annotate any function with `[grad]` to generate its derivative:

```
[grad]
fn loss(w: Matrix, x: Vector, y: Vector) → Float32 {
    let y_pred = x * w;
    sum((y - y_pred) ^ 2)
}

// Compiler generates:
fn loss_grad(w, x, y) → (Float32, Matrix) {
    let y_pred = x * w;
    let loss = sum((y - y_pred) ^ 2);
    let dl_dw = -2 * x^T * (y - y_pred);
    (loss, dl_dw)
}
```

## 6. Future Directions

The integration of AD with effect handlers (algebraic effects) enables differentiable programming with control flow: conditional branches, early returns, and exceptions can be differentiated through effect-aware AD transformations. The Koka language's work on differentiable programming with algebraic effects provides a template for this integration.

Another frontier is the verification of AD correctness. While AD is mathematically sound, implementation errors can produce incorrect gradients. Formal verification of AD transformations using theorem proving (e.g., in Coq or Lean) ensures that the generated gradient code is correct. Kasteran\*'s SMT-backed verification can check gradient correctness for individual computations.

## Works Cited

- Abadi, Martín, et al. “TensorFlow: A System for Large-Scale Machine Learning.” *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 265-283.
- Baur, Walter, and Volker Strassen. “The Complexity of Partial Derivatives.” *Theoretical Computer Science*, vol. 22, no. 3, 1983, pp. 317-330.
- Baydin, Atilim Gunes, et al. “Automatic Differentiation in Machine Learning: A Survey.” *Journal of Machine Learning Research*, vol. 18, no. 153, 2018, pp. 1-43.
- Bradbury, James, et al. “JAX: Composable Transformations of Python+NumPy Programs.” *GitHub*, 2018.
- Bücker, H. Martin, et al. *Automatic Differentiation: Applications, Theory, and Implementations*. Springer, 2006.
- Clifford, William Kingdon. “Preliminary Sketch of Biquaternions.” *Proceedings of the London Mathematical Society*, vol. 4, no. 1, 1873, pp. 381-395.
- Paszke, Adam, et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library.” *Advances in Neural Information Processing Systems 32*, 2019, pp. 8024-8035.
- Wengert, R. E. “A Simple Automatic Derivative Evaluation Program.” *Communications of the ACM*, vol. 7, no. 8, 1964, pp. 463-464.
- Griewank, Andreas, and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd ed., SIAM, 2008.
- Corliss, George, et al. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. Springer, 2002.
- Naumann, Uwe. *The Art of Differentiating Computer Programs*. SIAM, 2012.
- Griewank, Andreas. “On Automatic Differentiation.” *Mathematical Programming: Recent Developments and Applications*, 1989, pp. 83-107.
- Pearlmutter, Barak A., and Jeffrey Mark Siskind. “Reverse-Mode AD in a Functional Framework.” *Proceedings of the 2002 ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2002, pp. 1-12.
- Siskind, Jeffrey Mark, and Barak A. Pearlmutter. “Efficient Implementation of a Higher-Order Language with Built-In AD.” *Proceedings of the 2008 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, 2008, pp. 1-10.
- Elliott, Conal. “Beautiful Differentiation.” *Proceedings of the 2009 ACM SIGPLAN International Conference on Functional Programming*, 2009, pp. 1-12.
- Radul, Alexey, et al. “Theano: New Features and Speed Improvements.” *Proceedings of the 2012 Workshop on Machine Learning Systems*, 2012, pp. 1-10.
- Maclaurin, Dougal, et al. “Autograd: Effortless Gradients in NumPy.” *Proceedings of the 2015 International Conference on Machine Learning*, 2015, pp. 1-10.
- Innes, Michael, et al. “Differentiable Programming: A Survey.” *arXiv:1906.02924*, 2019.

van der Walt, Stefan, et al. “The NumPy Array: A Structure for Efficient Numerical Computation.” *Computing in Science and Engineering*, vol. 13, no. 2, 2011, pp. 22-30.

Oliphant, Travis E. *A Guide to NumPy*. CreateSpace, 2006.

Pasareanu, Corina, et al. “Symbolic Execution for Automatic Differentiation.” *Proceedings of the 2023 International Conference on Software Engineering*, 2023, pp. 1-14.

Liao, Hansheng, et al. “Differentiable Programming for Systems and Control.” *Proceedings of the IEEE*, vol. 109, no. 5, 2021, pp. 777-798.